UNIT – III

1.Analyze the MapReduce Failures. (12M)

Ans:

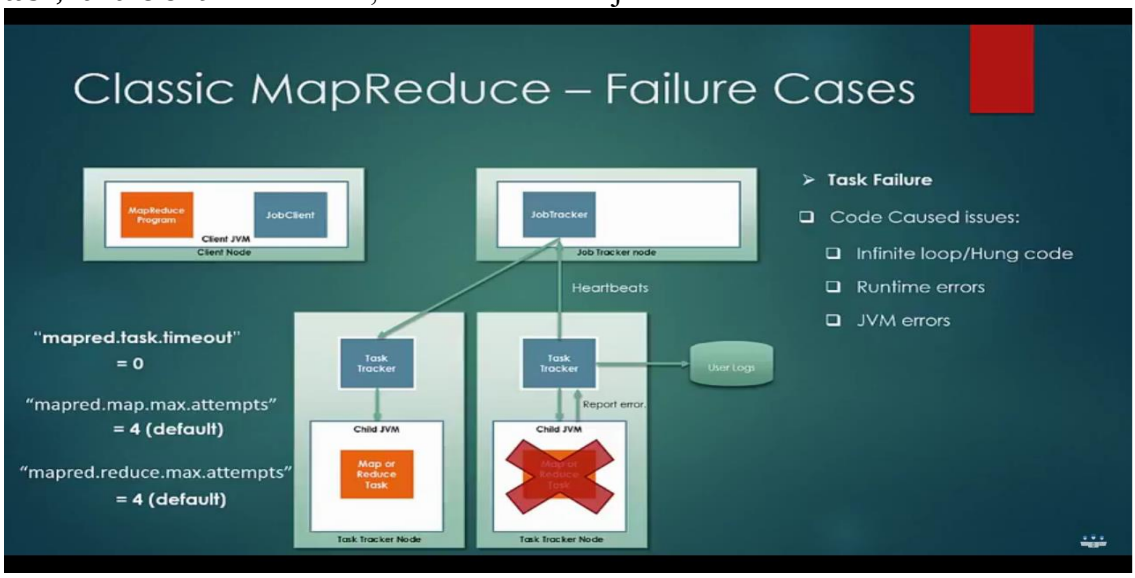
In the real world, user code is buggy, processes crash, and machines fail. One of the major benefits of using Hadoop is its ability to handle such failures and allow your job to complete successfully. We need to consider the failure of any of the following entities the task, the application master, the node manager, and the resource manager.

Failures

In the real world, user code is *buggy*, processes *crash*, and machines *fail*. One of the major benefits of using Hadoop is its ability to handle such *failures* and allow your job to complete.

Failures in Classic MapReduce

In the MapReduce 1 runtime there are three failure modes to consider: failure of the running task, failure of the tastracker, and failure of the jobtracker. Let's look at each in turn.



Task Failure

Consider first the case of the child task failing. The most common way that this happens is when user code in the map or reduce task throws a runtime exception. If this happens, the child JVM reports the error back to its parent tasktracker, before it exits. The error ultimately makes it into the user logs. The tasktracker marks the task attempt as *failed*, freeing up a slot to run another task.

For Streaming tasks, if the Streaming process exits with a nonzero exit code, it is marked as failed. This behavior is governed by the stream.non.zero.exit.is.failure property (the default is true).

Another failure mode is the sudden exit of the child JVM—perhaps there is a JVM bug that causes the JVM to exit for a particular set of circumstances exposed by the MapReduce user code. In this case, the tasktracker notices that the process has exited and marks the attempt as failed.

Hanging tasks are dealt with differently. The tasktracker notices that it hasn't received a progress update for a while and proceeds to mark the task as failed. The child JVM process will be automatically killed after this period. The timeout period after which tasks are considered failed is normally 10 minutes and can be configured on a per-job basis (or a cluster basis) by setting the mapred.task.timeout property to a value in milliseconds.

If a *Streaming* or *Pipes* process hangs, the tasktracker will kill it (along with the JVM that launched it) only in one the following circumstances: either mapred.task.tracker.task-controller is set to org.apache.hadoop.mapred.LinuxTaskController, or the default task controller in being used (org.apache.hadoop.mapred.DefaultTaskController) and the setsid command is available on the system (so that the child JVM and any processes it launches are in the same process group). In any other case orphaned Streaming or Pipes processes will accumulate on the system, which will impact utilization over time.

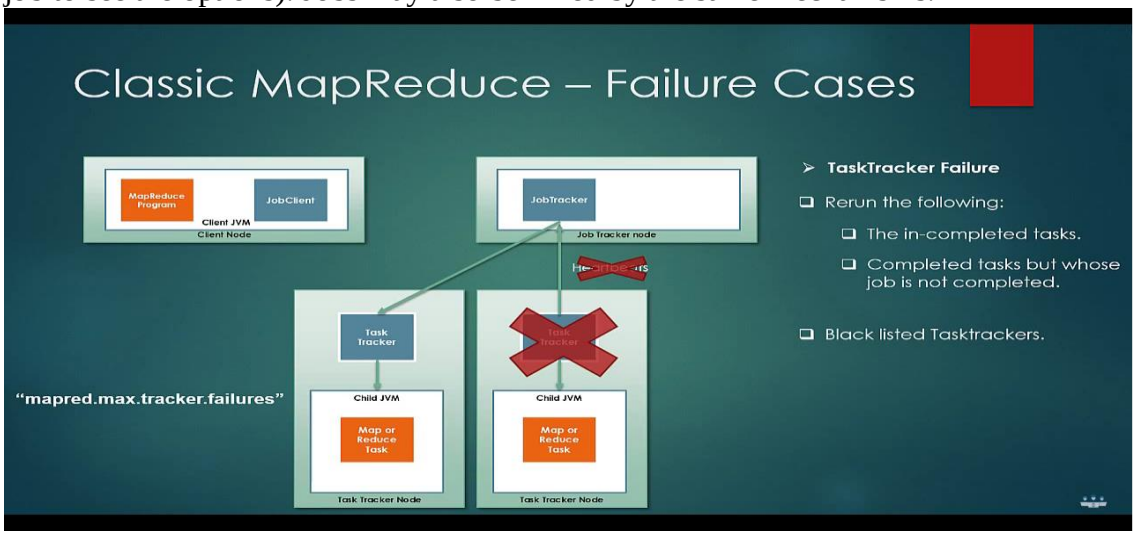
Setting the timeout to a value of zero disables the timeout, so long-running tasks are never marked as failed. In this case, a hanging task will never free up its slot, and over time there may be cluster slowdown as a result. This approach should therefore be avoided, and making sure that a task is reporting progress periodically will suffice.

When the jobtracker is notified of a task attempt that has failed (by the tasktracker's heartbeat call), it will reschedule execution of the task. The jobtracker will try to avoid rescheduling the task on a tasktracker where it has previously failed. Furthermore, if a task fails four times (or more), it will not be retried further. This value is configurable: the maximum number of attempts to run a task is controlled by the mapred.map.max.attempts property for map tasks and mapred.reduce.max.attempts for reduce tasks. By default, if any task fails four times (or whatever the maximum number of attempts is configured to), the whole job fails.

For some applications, it is undesirable to abort the job if a few tasks fail, as it may be possible to use the results of the job despite some failures. In this case, the maximum percentage of tasks that are allowed to fail without triggering job failure can be set for the job. Map tasks and reduce tasks are controlled independently, using the mapred.max.map.failures.percent and mapred.max.reduce.failures.percent properties.

A task attempt may also be *killed*, which is different from it failing. A task attempt may be killed because it is a speculative duplicate, or because the tasktracker it was running on failed, and the jobtracker marked all the task attempts running on it as killed. Killed task attempts do not count against the number of attempts to run the task (as set by mapred.map.max.attempts and mapred.reduce.max.attempts), since it wasn't the task's fault that an attempt was killed.

Users may also kill or fail task attempts using the web UI or the command line (type hadoop job to see the options). Jobs may also be killed by the same mechanisms.

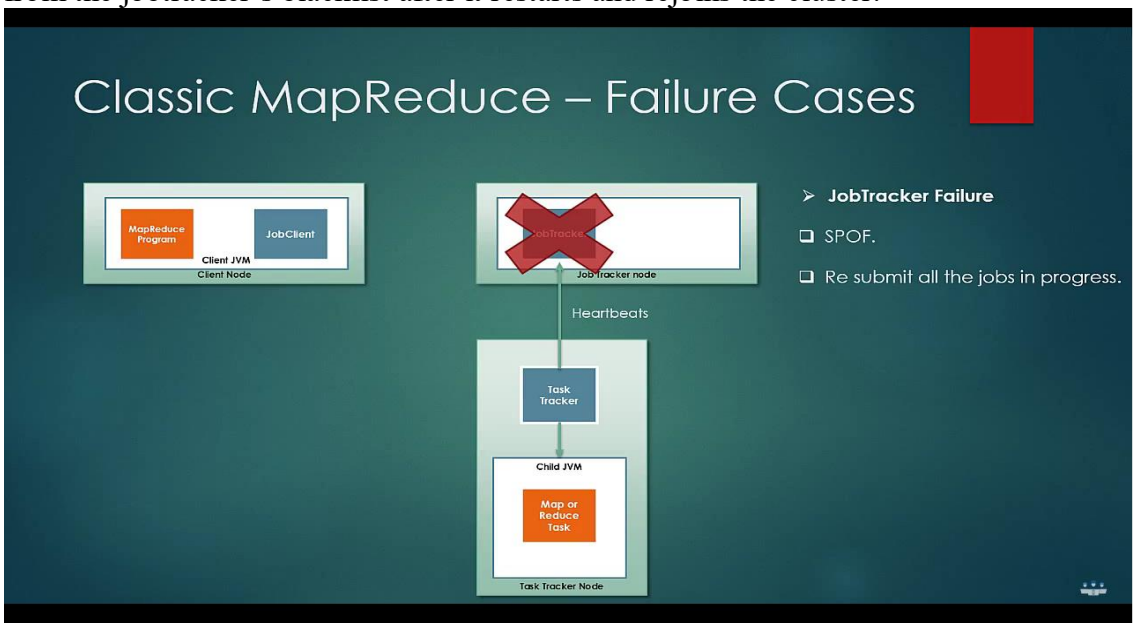


Tasktracker Failure

Failure of a tasktracker is another failure mode. If a tasktracker fails by crashing, or running very slowly, it will stop sending heartbeats to the jobtracker (or send them very infrequently). The jobtracker will notice a tasktracker that has stopped sending heartbeats (if it hasn't received one for 10 minutes, configured via the mapred.task tracker.expiry.interval property, in milliseconds) and remove it from its pool of tasktrackers to schedule tasks on. The jobtracker arranges for map tasks that were run and completed successfully on that tasktracker to be rerun if they belong to incomplete jobs, since their intermediate output residing on the failed tasktracker's local filesystem may not be accessible to the reduce task. Any tasks in progress are also rescheduled.

A tasktracker can also be *blacklisted* by the jobtracker, even if the tasktracker has not failed. If more than four tasks from the same job fail on a particular tasktracker (set by (mapred.max.tracker.failures), then the jobtracker records this as a fault. A tasktracker is blacklisted if the number of faults is over some minimum threshold (four, set by mapred.max.tracker.blacklists) and is significantly higher than the average number of faults for tasktrackers in the cluster cluster.

Blacklisted tasktrackers are not assigned tasks, but they continue to communicate with the jobtracker. Faults expire over time (at the rate of one per day), so tasktrackers get the chance to run jobs again simply by leaving them running. Alternatively, if there is an underlying fault that can be fixed (by replacing hardware, for example), the tasktracker will be removed from the jobtracker's blacklist after it restarts and rejoins the cluster.



Jobtracker Failure

Failure of the jobtracker is the most serious failure mode. Hadoop has no mechanism for dealing with failure of the jobtracker—it is a single point of failure—so in this case the job fails. However, this failure mode has a low chance of occurring, since the chance of a particular machine failing is low. The good news is that the situation is improved in YARN, since one of its design goals is to eliminate single points of failure in MapReduce.

After restarting a jobtracker, any jobs that were running at the time it was stopped will need to be re-submitted. There is a configuration option that attempts to recover any running jobs (mapred.jobtracker.restart.recover, turned off by default), however it is known not to work reliably, so should not be used.

2. List and explain various Hadoop data types(6M).

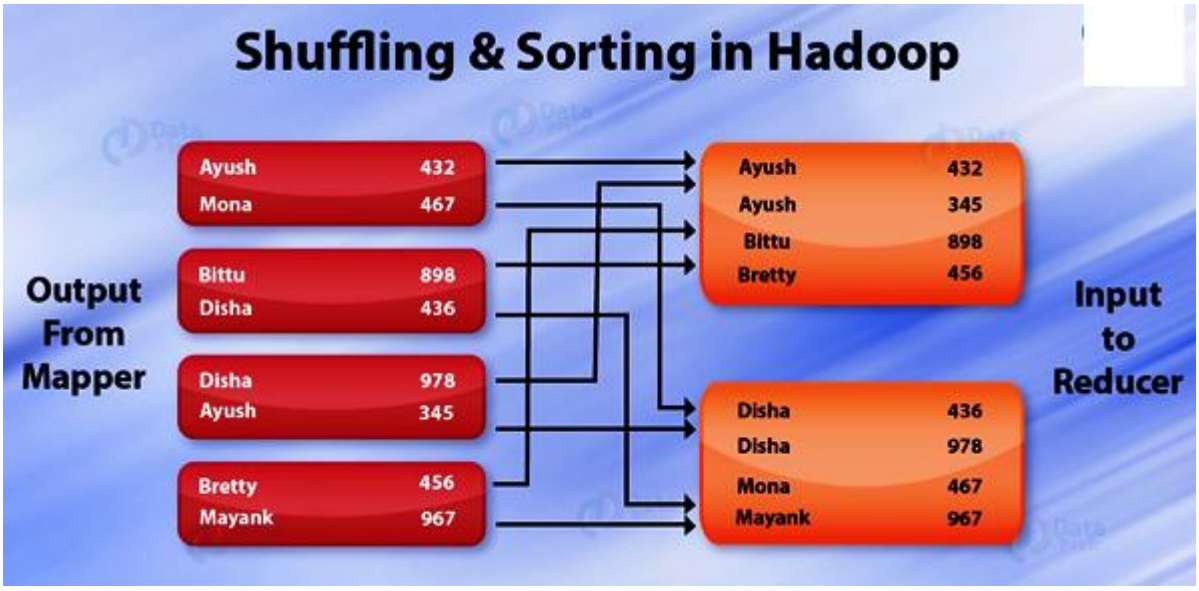
Ans:

the list of few data types in Java along with the equivalent Hadoop variant:

1. **Integer** –> **IntWritable**: It is the Hadoop variant of *Integer*. It is used to pass integer numbers as key or value.
2. **Float** –> **FloatWritable**: Hadoop variant of *Float* used to pass floating point numbers as key or value.
3. **Long** –> **LongWritable**: Hadoop variant of *Long* data type to store long values.
4. **Short** –> **ShortWritable**: Hadoop variant of *Short* data type to store short values.
5. **Double** –> **DoubleWritable**: Hadoop variant of *Double* to store double values.
6. **String** –> **Text**: Hadoop variant of *String to pass string characters as key or value.*
7. **Byte** –> **ByteWritable**: Hadoop variant of *byte* to store sequence of bytes.
8. **null** –> **NullWritable**: Hadoop variant of *null* to pass null as a key or value. Usually *NullWritable* is used as data type for output key of the reducer, when the output key is not important in the final result.

3. Explain the Shuffle and sort phase of MapReduce algorithm. (12M)

Ans:



The process of transferring data from the mappers to reducers is known as shuffling i.e. the process by which the system performs the sort and transfers the map output to the reducer as input. So, MapReduce shuffle phase is necessary for the reducers, otherwise, they would not have any input (or input from every mapper). As shuffling can start even before the map phase has finished so this saves some time and completes the tasks in lesser time.

Sorting in MapReduce

The keys generated by the mapper are automatically sorted by MapReduce Framework, i.e. Before starting of reducer, all intermediate **key-value pairs** in MapReduce that are generated by mapper get sorted by key and not by value. Values passed to each reducer are not sorted; they can be in any order.

Sorting in Hadoop helps reducer to easily distinguish when a new reduce task should start. This saves time for the reducer. Reducer starts a new reduce task when the next key in the sorted input data is different than the previous. Each reduce task takes key-value pairs as input and generates key-value pair as output.

Note that shuffling and sorting in Hadoop MapReduce is not performed at all if you specify zero reducers (setNumReduceTasks(0)). Then, the MapReduce job stops at the map phase, and the map phase does not include any kind of sorting (so even the map phase is faster).

Secondary Sorting in MapReduce

If we want to sort reducer's values, then the secondary sorting technique is used as it enables us to sort the values (in ascending or descending order) passed to each reducer.

Shuffling-Sorting occurs simultaneously to summarize the Mapper intermediate output. Shuffling and sorting in Hadoop MapReduce are not performed at all if you specify zero reducers (setNumReduceTasks(0)).

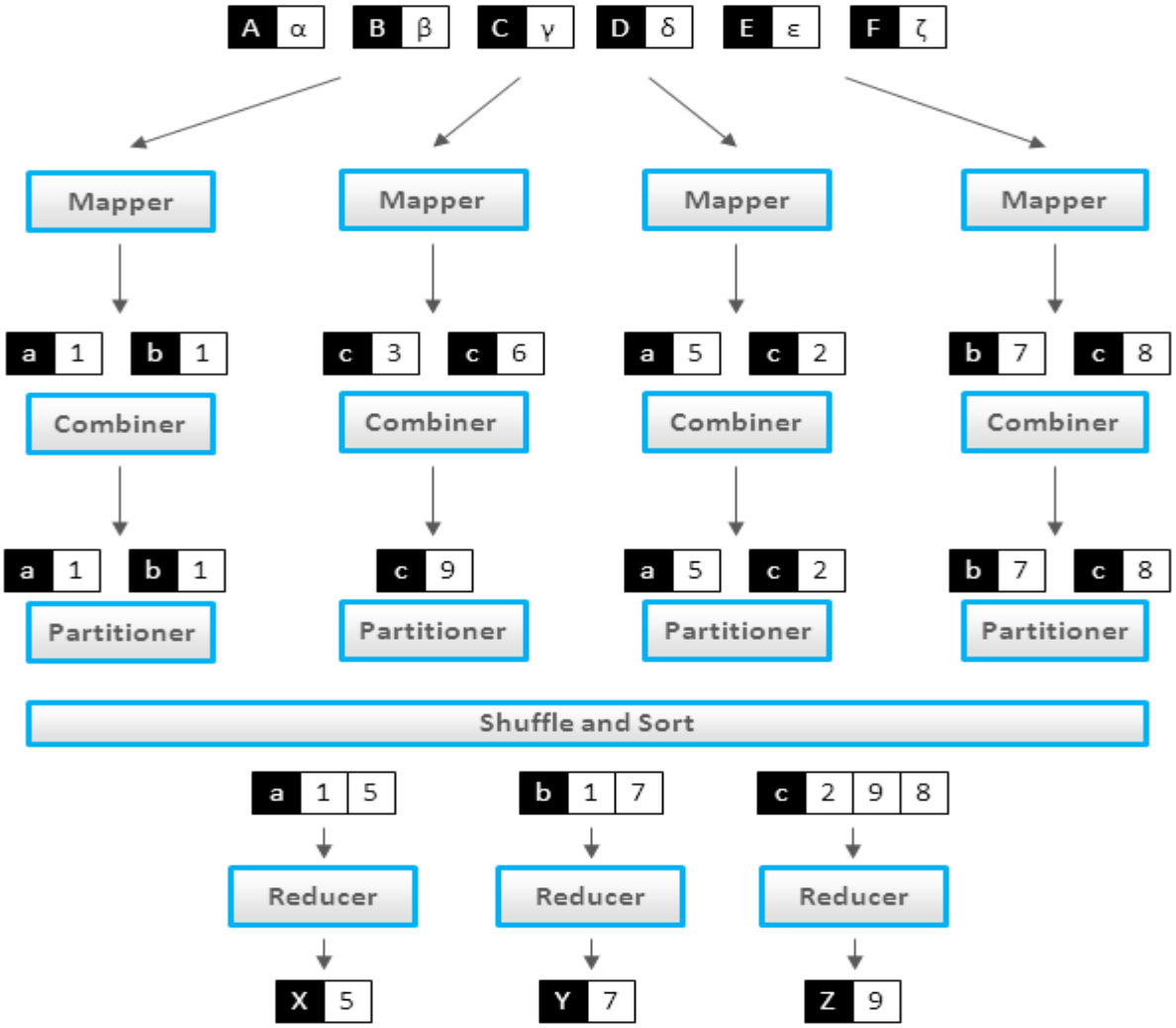
4. Anatomy of a MapReduce Job run in Apache Hadoop (12M)

Ans: Hadoop MapReduce jobs are divided into a set of map tasks and reduce tasks that run in a distributed fashion on a cluster of computers. Each task work on a small subset of the data it has been assigned so that the load is spread across the cluster.

The input to a MapReduce job is a set of files in the data store that are spread out over the HDFS. In Hadoop, these files are split with an input format, which defines how to separate a files into input split. You can assume that input split is a byte-oriented view of a chunk of the files to be loaded by a map task.

The map task generally performs loading, parsing, transformation and filtering operations, whereas reduce task is responsible for grouping and aggregating the data produced by map tasks to generate final output. This is the way a wide range of problems can be solved with such a straightforward paradigm, from simple numerical aggregation to complex join operations and cartesian products.

Each map task in Hadoop is broken into following phases: record reader, mapper, combiner, partitioner. The output of map phase, called intermediate key and values are sent to the reducers. The reduce tasks are broken into following phases: shuffle, sort, reducer and output format. The map tasks are assigned by Hadoop framework to those DataNodes where the actual data to be processed resides. This ensures that the data typically doesn't have to move over the network to save the network bandwidth and data is computed on the local machine itself so called map task is data local.



Mapper

Record Reader:

The record reader translates an input split generated by input format into records. The purpose of record reader is to parse the data into record but doesn't parse the record itself. It passes the data to the mapper in form of key/value pair. Usually the key in this context is positional information and the value is a chunk of data that composes a record. In our future articles we will discuss more about NLineInputFormat and custom record readers.

Map:

Map function is the heart of mapper task, which is executed on each key/value pair from the record reader to produce zero or more key/value pair, called intermediate pairs. The decision of what is key/value pair depends on what the MapReduce job is accomplishing. The data is grouped on key and the value is the information pertinent to the analysis in the reducer.

Combiner:

Its an optional component but highly useful and provides extreme performance gain of MapReduce job without any downside. Combiner is not applicable to all the MapReduce algorithms but where ever it can be applied it is always recommended to use. It takes the intermediate keys from the mapper and applies a user-provided method to aggregate values in a small scope of that one mapper. e.g sending (hadoop, 3) requires fewer bytes than sending (hadoop, 1) three times over the network. We will cover combiner in much more depth in our future articles.

Partitioner:

The partitioner takes the intermediate key/value pairs from mapper and split them into shards, one shard per reducer. This randomly distributes the keyspace evenly over the reducer, but still ensures that keys with the same value in different mappers end up at the same reducer. The partitioned data is written to the local filesystem for each map task and waits to be pulled by its respective reducer.

Reducer

Shuffle and Sort:

The reduce task start with the shuffle and sort step. This step takes the output files written by all of the partitioners and downloads them to the local machine in which the reducer is running. These individual data pipes are then sorted by keys into one larger data list. The purpose of this sort is to group equivalent keys together so that their values can be iterated over easily in the reduce task.

Reduce:

The reducer takes the grouped data as input and runs a reduce function once per key grouping. The function is passed the key and an iterator over all the values associated with that key. A wide range of processing can happen in this function, the data can be aggregated, filtered, and combined in a number of ways. Once it is done, it sends zero or more key/value pair to the final step, the output format.

Output Format:

The output format translate the final key/value pair from the reduce function and writes it out to a file by a record writer. By default, it will separate the key and value with a tab and separate record with a new line character. We will discuss in our future articles about how to write your own customized output format.

5. Explain the scheduling in MapReduce.(12M)

Ans : In Hadoop 2, a **YARN** called Yet Another Resource Negotiator was introduced. The basic idea behind the YARN introduction is to split the functionalities of resource management and job scheduling or monitoring into separate daemons that are ResorceManager, Application Master, and NodeManager.

ResorceManager is the master daemon that arbitrates resources among all the applications in the system. NodeManager is the slave daemon responsible for containers, monitoring their resource usage, and reporting the same to ResourceManager or Schedulers. ApplicationMaster negotiates resources from the ResourceManager and works with NodeManager in order to execute and monitor the task.

The ResourceManager has two main components that are Schedulers and Applications Manager. **Schedulers** in YARN ResourceManager is a pure scheduler which is responsible for allocating resources to the various running applications. It is not responsible for monitoring or tracking the status of an application. Also, the scheduler does not guarantee about restarting the tasks that are failed either due to hardware failure or application failure.

The scheduler performs scheduling based on the resource requirements of the applications.

It has some pluggable policies that are responsible for partitioning the cluster resources among the various queues, applications, etc.

The FIFO Scheduler, CapacityScheduler, and FairScheduler are such pluggable policies that are responsible for allocating resources to the applications.
Let us now study each of these Schedulers in detail.

TYPES OF HADOOP SCHEDULER

1. FIFO Scheduler

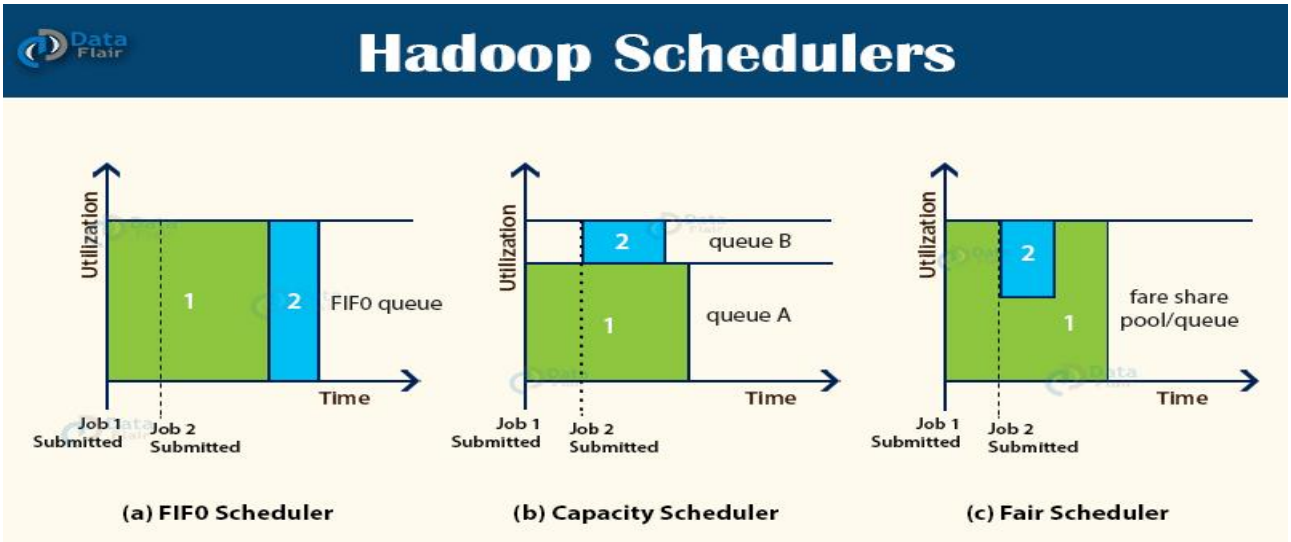
First In First Out is the default scheduling policy used in Hadoop. FIFO Scheduler gives more preferences to the application coming first than those coming later. It places the applications in a queue and executes them in the order of their submission (first in, first out). Here, irrespective of the size and priority, the request for the first application in the queue are allocated first. Once the first application request is satisfied, then only the next application in the queue is served.

Advantage:

- It is simple to understand and doesn't need any configuration.
- Jobs are executed in the order of their submission.

Disadvantage:

- It is not suitable for shared clusters. If the large application comes before the shorter one, then the large application will use all the resources in the cluster, and the shorter application has to wait for its turn. This leads to starvation.
- It does not take into account the balance of resource allocation between the long applications and short applications.



2. Capacity Scheduler

The CapacityScheduler allows multiple-tenants to securely share a large Hadoop cluster. It is designed to run Hadoop applications in a shared, multi-tenant cluster while maximizing the throughput and the utilization of the cluster.

It supports hierarchical queues to reflect the structure of organizations or groups that utilizes the cluster resources. A queue hierarchy contains three types of queues that are root, parent, and leaf.

The root queue represents the cluster itself, parent queue represents organization/group or sub-organization/sub-group, and the leaf accepts application submission.

The Capacity Scheduler allows the sharing of the large cluster while giving capacity guarantees to each organization by allocating a fraction of cluster resources to each queue.
Also, when there is a demand for the free resources that are available on the queue who has completed its task, by the queues running below capacity, then these resources will be assigned to the applications on queues running below capacity. This provides elasticity for the organization in a cost-effective manner.
Apart from it, the CapacityScheduler provides a comprehensive set of limits to ensure that a single application/user/queue cannot use a disproportionate amount of resources in the cluster.
To ensure fairness and stability, it also provides limits on initialized and pending apps from a single user and queue.

Advantages:

- It maximizes the utilization of resources and throughput in the Hadoop cluster.
- Provides elasticity for groups or organizations in a cost-effective manner.
- It also gives capacity guarantees and safeguards to the organization utilizing cluster.

Disadvantage:

- It is complex amongst the other scheduler.

3. Fair Scheduler

FairScheduler allows YARN applications to fairly share resources in large Hadoop clusters. With FairScheduler, there is no need for reserving a set amount of capacity because it will dynamically balance resources between all running applications.
It assigns resources to applications in such a way that all applications get, on average, an equal amount of resources over time.
The FairScheduler, by default, takes scheduling fairness decisions only on the basis of memory. We can configure it to schedule with both memory and CPU.
When the single application is running, then that app uses the entire cluster resources. When other applications are submitted, the free up resources are assigned to the new apps so that every app eventually gets roughly the same amount of resources. FairScheduler enables short apps to finish in a reasonable time without starving the long-lived apps.
Similar to CapacityScheduler, the FairScheduler supports hierarchical queue to reflect the structure of the long shared cluster.
Apart from fair scheduling, the FairScheduler allows for assigning minimum shares to queues for ensuring that certain users, production, or group applications always get sufficient resources. When an app is present in the queue, then the app gets its minimum share, but when the queue doesn't need its full guaranteed share, then the excess share is split between other running applications.

Advantages:

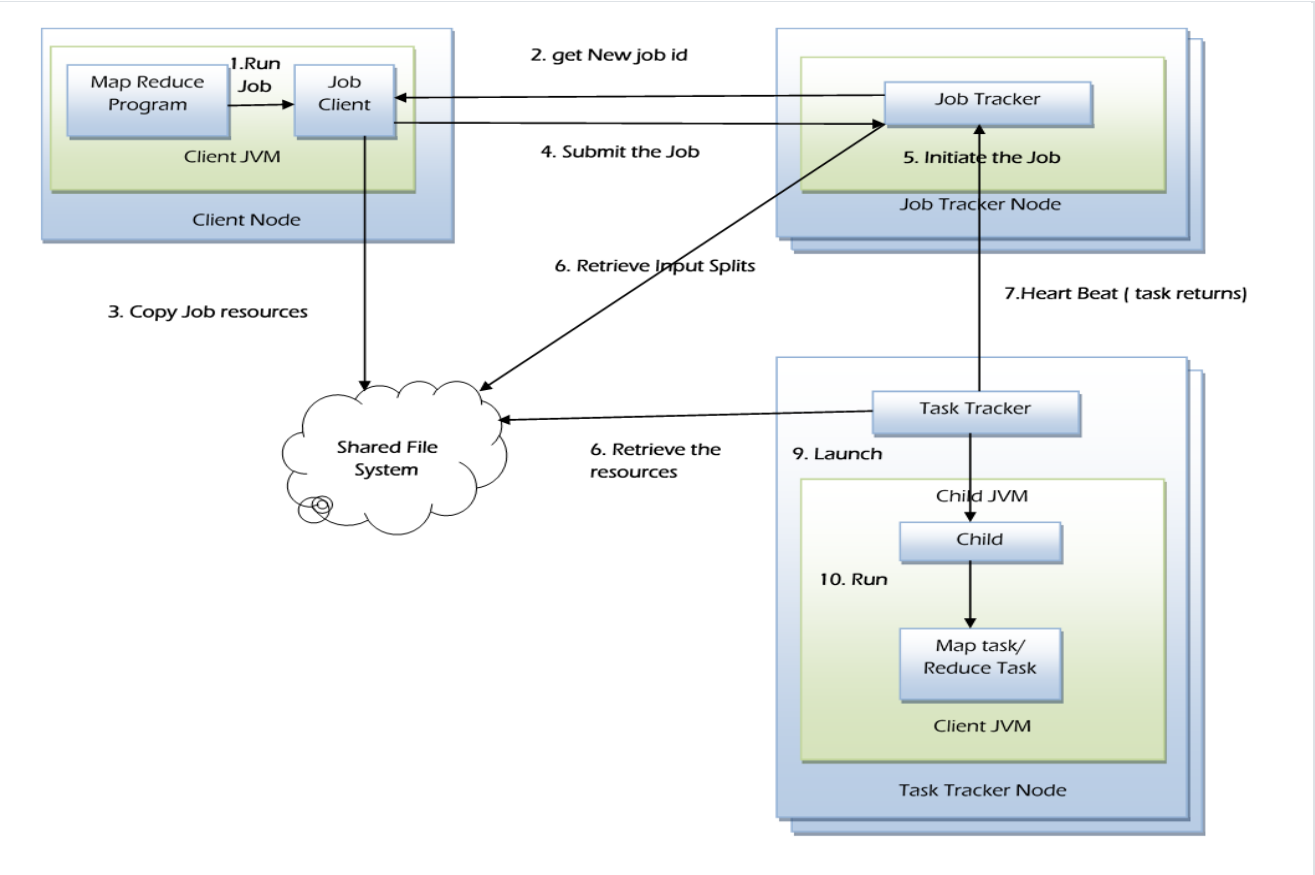
- It provides a reasonable way to share the Hadoop Cluster between the number of users.
- Also, the FairScheduler can work with app priorities where the priorities are used as weights in determining the fraction of the total resources that each application should get.

Disadvantage:

- It requires configuration.

6. MapReduce - Job Execution (or) Speculative MapReduce - Job Execution Execution process (12M)

Ans:



MapReduce is a programming model designed to process large amount of data in parallel by dividing the job into several independent local tasks.
Running the independent tasks locally reduces the network usage drastically.
To run the tasks locally, the data needs move to the data nodes for data processing.
The below tasks occur when the user submits a MapReduce job to Hadoop -

- The local Job Client prepares the job for submission and hands it off to the Job Tracker.
- The Job Tracker schedules the job and distributes the map work among the Task Trackers for parallel processing.
- Each Task Tracker issues a Map Task.
- The Job Tracker receives progress information from the Task Trackers.
- Once the mapping phase results available, the Job Tracker distributes the reduce work among the Task Trackers for parallel processing.
- Each Task Tracker issues a Reduce Task to perform the work.

- The Job Tracker receives progress information from the Task Trackers.
- Once the Reduce task completed, Cleanup task will be performed.

The following are the main phases in map reduce job execution flow -

- Job Submission
- Job Initialization
- Task Assignment
- Task Execution
- Job/Task Progress
- Job Completion

Job Submission: -

The job **submit** method creates an internal instance of **JobSubmitter** and calls **submitJobInternal** method on it.
waitForCompletion method samples the job's progress once a second after the job submitted.
waitForCompletion method performs below -

- It goes to Job Tracker and gets a jobId for the job
- Perform checks if the output directory has been specified or not.
- If specified checks the directory already exists or is new and throws error if any issue occurs with directory.
- Computes input split and throws error if it fails because the input paths don't exist.
- Copies the resources to Job Tracker file system in a directory named after Job Id.
- Finally, it calls **submitJob** method on JobTracker.

Job Initialization: -
Job tracker performs the below steps in job initialization -

- Creates object to track tasks and their progress.
- Creates a map tasks for each input split.
- The number of reduce tasks is defined by the configuration **mapred.reduce.tasks** set by **setNumReduceTasks** method.
- Tasks are assigned with task ID's.
- Job initialization task and Job clean up task created and these are run by task trackers.
- Job clean up tasks which delete the temporary directory after the job is completed.

Task Assignment: -
Task Tracker sends a heartbeat to job tracker every five seconds.
The heartbeat is a communication channel and indicate whether it is ready to run a new task.
The available slots information also sends to them.
The job allocation takes place like below -

- Job Tracker first selects a job to select the task based on job scheduling algorithms.
- The default scheduler fills empty map task before reduce task slots.
- The number of slots which a task tracker has depends on number of cores.

Task Execution: -
Below steps describes how the job executed -

- Task Tracker copies the job jar file from the shared filesystem (HDFS).

- Task Tracker creates a local working directory and un-jars the jar file into the local file system.
- Task Tracker creates an instance of TaskRunner.
- Task Tracker starts TaskRunner in a new JVM to run the map or reduce task.
- The child process communicates the progress to parent process.
- Each task can perform setup and cleanup actions based on **OutputComitter**.
- The input provided via stdin and get output via stdout from the running process even if the map or reduce tasks ran via pipes or socket in case of streaming.

Job/Task Progress: -

Below steps describes about how the progress is monitored of a job/task -

- Job Client keeps polling the Job Tracker for progress.
- Each child process reports its progress to parent task tracker.
- If a task reports progress, it sets a flag to indicate the status change that sent to the Task Tracker.
- The flag is verified in a separate thread for every 3 seconds and it notifies the Task Tracker of the current task status if set.
- Task tracker sends its progress to Job Tracker over the heartbeat for every five seconds.
- Job Tracker consolidate the task progress from all task trackers and keeps a holistic view of job.
- The Job receives the latest status by polling the Job Tracker every second.

Job Completion: -

Once the job completed, the clean-up task will get processed like below -

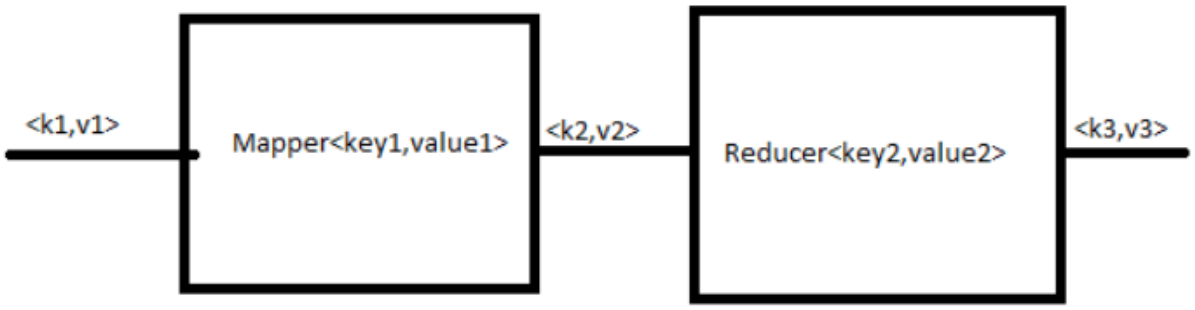
- Once the task completed, Task Tracker sends the job completion status to the Job Tracker.
- Job Tracker then send the job completion message to client.
- Job Tracker cleans up its current working state for the job and instructs Task Trackers to perform the same and also cleans up all the temporary directories.
- The total process causes Job Client's **waitForJobToComplete** method to return

7. Map Reduce Types and Formats (12M)

Ans:

- MapReduce types and Formats: MapReduce has simple model of data processing: inputs and outputs for the map and reduce function are key-value pairs.
- MapReduce Types:
  map: (K1, V1) → list(K2, V2)
  reduce: (K2, list(V2)) → list(K3, V3)

In general, map input key and value types(K1 and V1) are different from the map output types(K2 and V2). However, reduce input must have same types as map output, although reduce output types may be different again(K3 and V3).




```
public class Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {
  public class Context extends MapContext<KEYIN, VALUEIN, KEYOUT, VALUEOUT>
{
    // ...
  }
  protected void map(KEYIN key, VALUEIN value,
                     Context context) throws IOException, InterruptedException {
    // ...
  }
}
public class Reducer<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {
  public class Context extends ReducerContext<KEYIN, VALUEIN, KEYOUT,
VALUEOUT> {
    // ...
  }
  protected void reduce(KEYIN key, Iterable<VALUEIN> values,

                        Context context) throws IOException, InterruptedException {
    // ...
  }
}
```

Context objects are used for emitting key-value pairs, so they are parameterized by output types, so that signature of the write() method is:

```
public void write(KEYOUT key, VALUEOUT value)
    throws IOException, InterruptedException
```

- Since Mapper and Reducer are separate classes the type parameters have different scopes and the actual type argument of KEYIN(say) in the Mapper may be different to the type of type parameter of same name(KEYIN) in the Reducer.
- In maximum temperature example, KEYIN is replaced by LongWritable for the Mapper, and by Text for the Reducer.
- Even though map output types and reduce input types must match, this is not enforced by the Java compiler.
- If combine function is used, then it is same form as the reduce function, except its output types are intermediate key and value types(K2 and V2) so they can feed the reduce function:

map: (K1, V1) → list(K2, V2)

combine: (K2, list(V2)) → list(K2, V2)
reduce: (K2, list(V2)) → list(K3, V3)

- Often the combine and reduce functions are the same, in which case, K3 is same as K2 and V3 is same as V2.

- Partition function operates on intermediate key and value types(K2 and V2), and returns the partition index. In practice, partition is determined solely by the key(value is ignored):

partition: (K2, V2) → integer

- The Partitioner in MapReduce controls the partitioning of the key of the intermediate mapper output.
- The partition function operates on the intermediate key and value types (K2 and V2) and returns the partition index.
- partition: (K2, V2) → integer
- public abstract class Partitioner<KEY, VALUE>

{
public abstract int getPartition(KEY key, VALUE value, int numPartitions);
}

Input Formats

Hadoop can process many different types of data formats, from flat text files to databases.

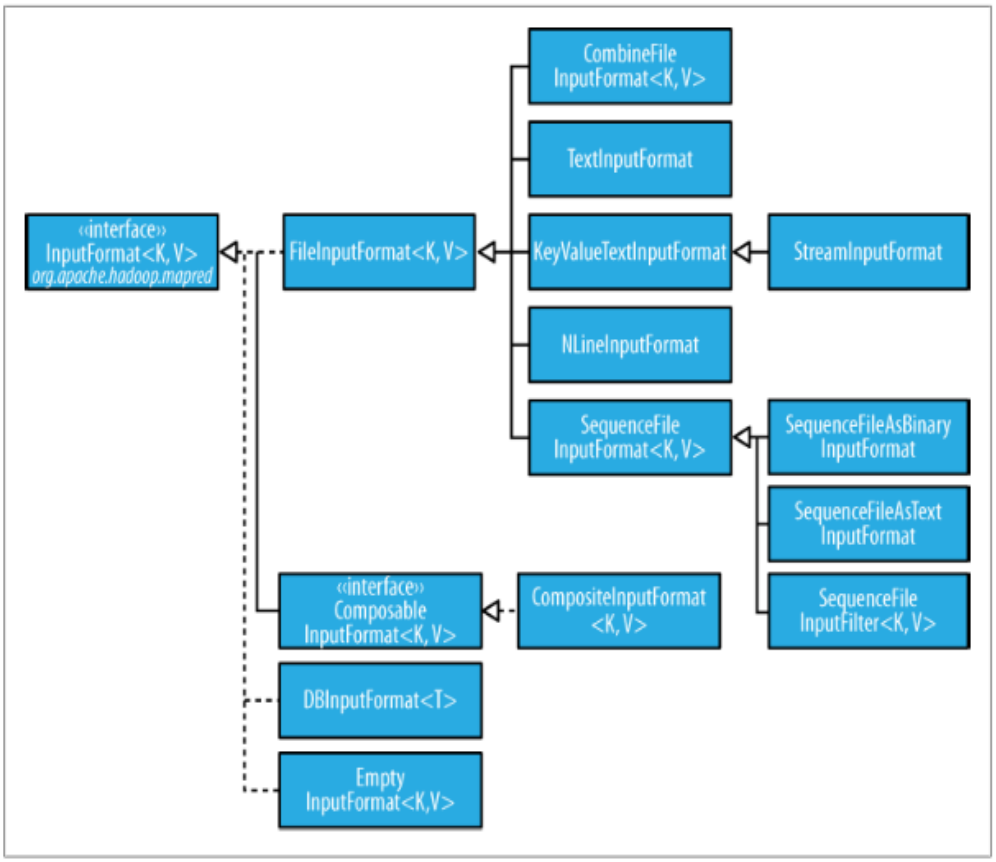


Figure 7-2. InputFormat class hierarchy

FileInputFormat

FileInputFormat is the base class for all implementations of InputFormat that use files as their data source. It provides two things: a place to define which files are included as the input to a job, and an implementation for generating splits for the input files. The job of dividing splits into records is performed by subclasses.

FileInputFormat input paths

The input to a job is specified as a collection of paths, which offers great flexibility in constraining the input. FileInputFormat offers four static convenience methods for setting a Job's input paths:

public static void addInputPath(Job job, Path path)
public static void addInputPaths(Job job, String commaSeparatedPaths)
public static void setInputPaths(Job job, Path... inputPaths)
public static void setInputPaths(Job job, String commaSeparatedPaths)

The addInputPath() and addInputPaths() methods add a path or paths to the list of inputs. You can call these methods repeatedly to build the list of paths. The setInputPaths() methods set the entire list of paths in one go (replacing any paths set on the Job in previous calls).

A path may represent a file, a directory, or, by using a glob, a collection of files and directories. A path representing a directory includes all the files in the directory as input to the job.

FileInputFormat input splits

Given a set of files, how does FileInputFormat turn them into splits? FileInputFormat splits only large files — here, "large" means larger than an HDFS block. The split size is normally the size of an HDFS block, which is appropriate for most applications; however, it is possible to control this value by setting various Hadoop properties, as shown in Table

Table 7-5. Properties for controlling split size

Property name	Type	Default value	Description
mapred.min.split.size	int	1	The smallest valid size in bytes for a file split.
mapred.max.split.size[a]	long	Long.MAX_VALUE, that is 9223372036854775807	The largest valid size in bytes for a file split.
dfs.block.size	long	64 MB, that is 67108864	The size of a block in HDFS in bytes.

The minimum split size is usually 1 byte, although some formats have a lower bound on the split size. (For example, sequence files insert sync entries every so often in the stream, so the minimum split size has to be large enough to ensure that every split has a sync point to allow the reader to resynchronize with a record boundary.)

Applications may impose a minimum split size. By setting this to a value larger than the block size, they can force splits to be larger than a block. There is no good reason for doing this when using HDFS, because doing so will increase the number of blocks that are not local to a map task.

The maximum split size defaults to the maximum value that can be represented by a Java long type. It has an effect only when it is less than the block size, forcing splits to be smaller than a block.

Small files and CombineFileInputFormat

Hadoop works better with a small number of large files than a large number of small files. One reason for this is that FileInputFormat generates splits in such a way that each split is all or part of a single file. If the file is very small ("small" means significantly smaller than an HDFS block) and there are a lot of them, each map task will process very little input, and there will be a lot of them (one per file), each of which imposes extra bookkeeping overhead. Compare a 1 GB file broken into eight 128 MB blocks with 10,000 or so 100 KB files. The 10,000 files use one map each, and the job time can be tens or hundreds of times slower than the equivalent one with a single input file and eight map tasks.

The situation is alleviated somewhat by CombineFileInputFormat, which was designed to work well with small files. Where FileInputFormat creates a split per file, CombineFileInputFormat packs many files into each split so that each mapper has more to process. Crucially, CombineFileInputFormat takes node and rack locality into account when deciding which blocks to place in the same split, so it does not compromise the speed at which it can process the input in a typical MapReduce job.

Text Input

Hadoop excels at processing unstructured text.

TextInputFormat

TextInputFormat is the default InputFormat. Each record is a line of input. The key, a LongWritable, is the byte offset within the file of the beginning of the line. The value is the contents of the line, excluding any line terminators (e.g., newline or carriage return), and is packaged as a Text object. So, a file containing the following text:

On the top of the Crumpetty Tree
The Quangle Wangle sat,
But his face you could not see,
On account of his Beaver Hat.

is divided into one split of four records. The records are interpreted as the following keyvalue pairs:

(0, On the top of the Crumpetty Tree)
(33, The Quangle Wangle sat,)
(57, But his face you could not see,)
(89, On account of his Beaver Hat.)

Clearly, the keys are not line numbers. This would be impossible to implement in general, in that a file is broken into splits at byte, not line, boundaries. Splits are processed independently. Line numbers are really a sequential notion. You have to keep a count of lines as you consume them, so knowing the line number within a split would be possible, but not within the file.

However, the offset within the file of each line is known by each split independently of the other splits, since each split knows the size of the preceding splits and just adds this onto the offsets within the split to produce a global file offset. The offset is usually sufficient for applications that need a unique identifier for each line. Combined with the file's name, it is unique within the filesystem. Of course, if all the lines are a fixed width, calculating the line number is simply a matter of dividing the offset by the width.

KeyValueTextInputFormat

TextInputFormat's keys, being simply the offsets within the file, are not normally very useful. It is common for each line in a file to be a key-value pair, separated by a delimiter such as a tab character. For example, this is the kind of output produced by

TextOutputFormat, Hadoop's default OutputFormat. To interpret such files correctly, KeyValueTextInputFormat is appropriate. You can specify the separator via the mapreduce.input.keyvaluelinerecordreader.key.value.separator property. It is a tab character by default. Consider the following input file, where → represents a (horizontal) tab character:

line1→On the top of the Crumpetty Tree
line2→The Quangle Wangle sat,
line3→But his face you could not see,
line4→On account of his Beaver Hat.

Like in the TextInputFormat case, the input is in a single split comprising four records, although this time the keys are the Text sequences before the tab in each line:

(line1, On the top of the Crumpetty Tree)
(line2, The Quangle Wangle sat,)
(line3, But his face you could not see,)
(line4, On account of his Beaver Hat.)

NLineInputFormat

With TextInputFormat and KeyValueTextInputFormat, each mapper receives a variable number of lines of input. The number depends on the size of the split and the length of the lines. If you want your mappers to receive a fixed number of lines of input, then NLineInputFormat is the InputFormat to use. Like with TextInputFormat, the keys are the byte offsets within the file and the values are the lines themselves.

N refers to the number of lines of input that each mapper receives. With N set to 1 (the default), each mapper receives exactly one line of input. The mapreduce.input.lineinputformat.linespermap property controls the value of N. By way of example, consider these four lines again:

On the top of the Crumpetty Tree
The Quangle Wangle sat,
But his face you could not see,
On account of his Beaver Hat.

If, for example, N is 2, then each split contains two lines. One mapper will receive the first two key-value pairs:

(0, On the top of the Crumpetty Tree)
(33, The Quangle Wangle sat,)

And another mapper will receive the second two key-value pairs:

(57, But his face you could not see,)
(89, On account of his Beaver Hat.)

The keys and values are the same as those that TextInputFormat produces. The difference is in the way the splits are constructed.

Binary Input

Hadoop MapReduce is not restricted to processing textual data. It has support for binary formats, too.

SequenceFileInputFormat

Hadoop's sequence file format stores sequences of binary key-value pairs. Sequence files are well suited as a format for MapReduce data because they are splittable (they have sync points so that readers can synchronize with record boundaries from an arbitrary point in the file, such as the start of a split), they support compression as a part of the format, and they can store arbitrary types using a variety of serialization frameworks.

To use data from sequence files as the input to MapReduce, you can use SequenceFileInputFormat. The keys and values are determined by the sequence file, and you need to make sure that your map input types correspond.

SequenceFileAsTextInputFormat

SequenceFileAsTextInputFormat is a variant of SequenceFileInputFormat that converts the sequence file's keys and values to Text objects. The conversion is performed by calling toString() on the keys and values. This format makes sequence files suitable input for Streaming.

SequenceFileAsBinaryInputFormat

SequenceFileAsBinaryInputFormat is a variant of SequenceFileInputFormat that retrieves the sequence file's keys and values as opaque binary objects. They are encapsulated as BytesWritable objects, and the application is free to interpret the underlying byte array as it pleases. In combination with a process that creates sequence files with SequenceFile.Writer's appendRaw() method or SequenceFileAsBinaryOutputFormat, this provides a way to use any binary data types with MapReduce (packaged as a sequence file), although plugging into Hadoop's serialization mechanism is normally a cleaner alternative.

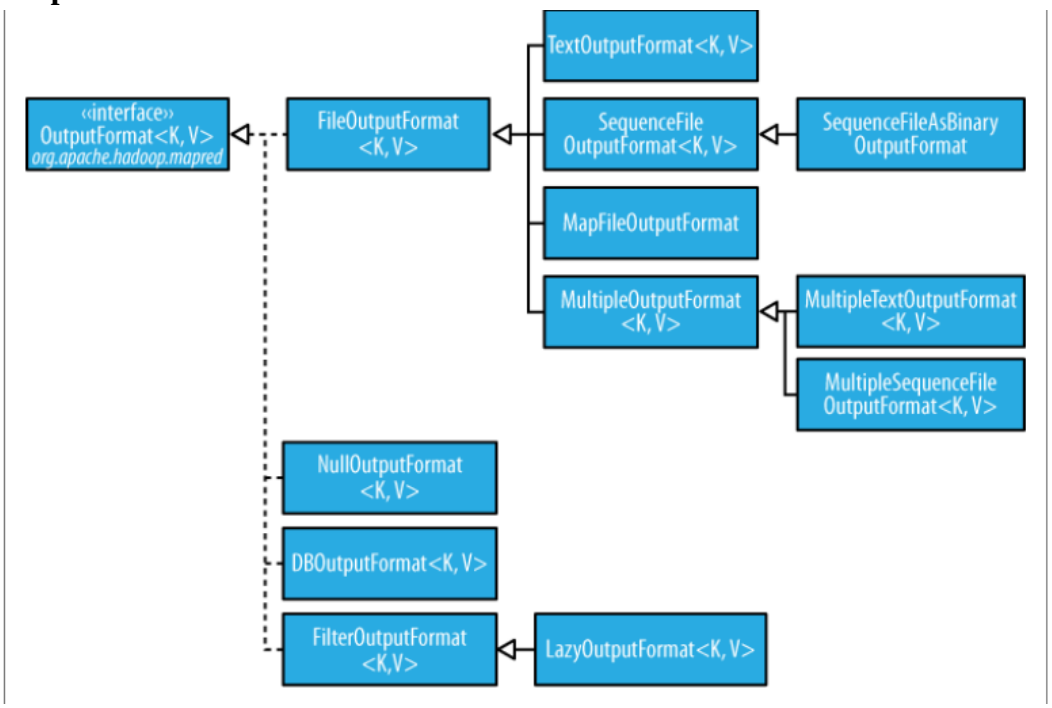
FixedLengthInputFormat

FixedLengthInputFormat is for reading fixed-width binary records from a file, when the records are not separated by delimiters. The record size must be set via fixedlengthinputformat.record.length.

Database Input (and Output)

DBInputFormat is an input format for reading data from a relational database, using JDBC. Because it doesn't have any sharding capabilities, you need to be careful not to overwhelm the database from which you are reading by running too many mappers. For this reason, it is best used for loading relatively small datasets, perhaps for joining with larger datasets from HDFS using MultipleInputs. The corresponding output format is DBOutputFormat, which is useful for dumping job outputs (of modest size) into a database.

Output Formats



Text Output

The default output format, TextOutputFormat, writes records as lines of text. Its keys and values may be of any type, since TextOutputFormat turns them to strings by calling toString() on them. Each key-value pair is separated by a tab character, although that may be changed using the mapreduce.output.textoutputformat.separator property. The counterpart to TextOutputFormat for reading in this case is KeyValueTextInputFormat, since it breaks lines into key-value pairs based on a configurable separator.

You can suppress the key or the value from the output (or both, making this output format equivalent to NullOutputFormat, which emits nothing) using a NullWritable type. This also causes no separator to be written, which makes the output suitable for reading in using TextInputFormat.

Binary Output

SequenceFileOutputFormat

As the name indicates, SequenceFileOutputFormat writes sequence files for its output. This is a good choice of output if it forms the input to a further MapReduce job, since it is compact and is readily compressed. Compression is controlled via the static methods on SequenceFileOutputFormat,

SequenceFileAsBinaryOutputFormat

SequenceFileAsBinaryOutputFormat — the counterpart to SequenceFileAsBinaryInputFormat — writes keys and values in raw binary format into a sequence file container.

MapFileOutputFormat

MapFileOutputFormat writes map files as output. The keys in a MapFile must be added in order, so you need to ensure that your reducers emit keys in sorted order.

Multiple Outputs

FileOutputFormat and its subclasses generate a set of files in the output directory. There is one file per reducer, and files are named by the partition number: *part-r-00000*, *part-r-00001*, and so on. Sometimes there is a need to have more control over the naming of the files or to produce multiple files per reducer.

MultipleOutputs allows you to write data to files whose names are derived from the output keys and values, or in fact from an arbitrary string. This allows each reducer (or mapper in a map-only job) to create more than a single file. Filenames are of the form *name-m-nnnnn* for map outputs and *name-r-nnnnn* for reduce outputs, where *name* is an arbitrary name that is set by the program and *nnnnn* is an integer designating the part number, starting from *00000*. The part number ensures that outputs written from different partitions (mappers or reducers) do not collide in the case of the same name.

Lazy Output

FileOutputFormat subclasses will create output (*part-r-nnnnn*) files, even if they are empty. Some applications prefer that empty files not be created, which is where LazyOutputFormat helps. It is a wrapper output format that ensures that the output file is created only when the first record is emitted for a given partition. To use it, call its setOutputFormatClass() method with the JobConf and the underlying output format. Streaming supports a -lazyOutput option to enable LazyOutputFormat.

Database Output

The output formats for writing to relational databases and to HBase are mentioned in Database Input (and Output).

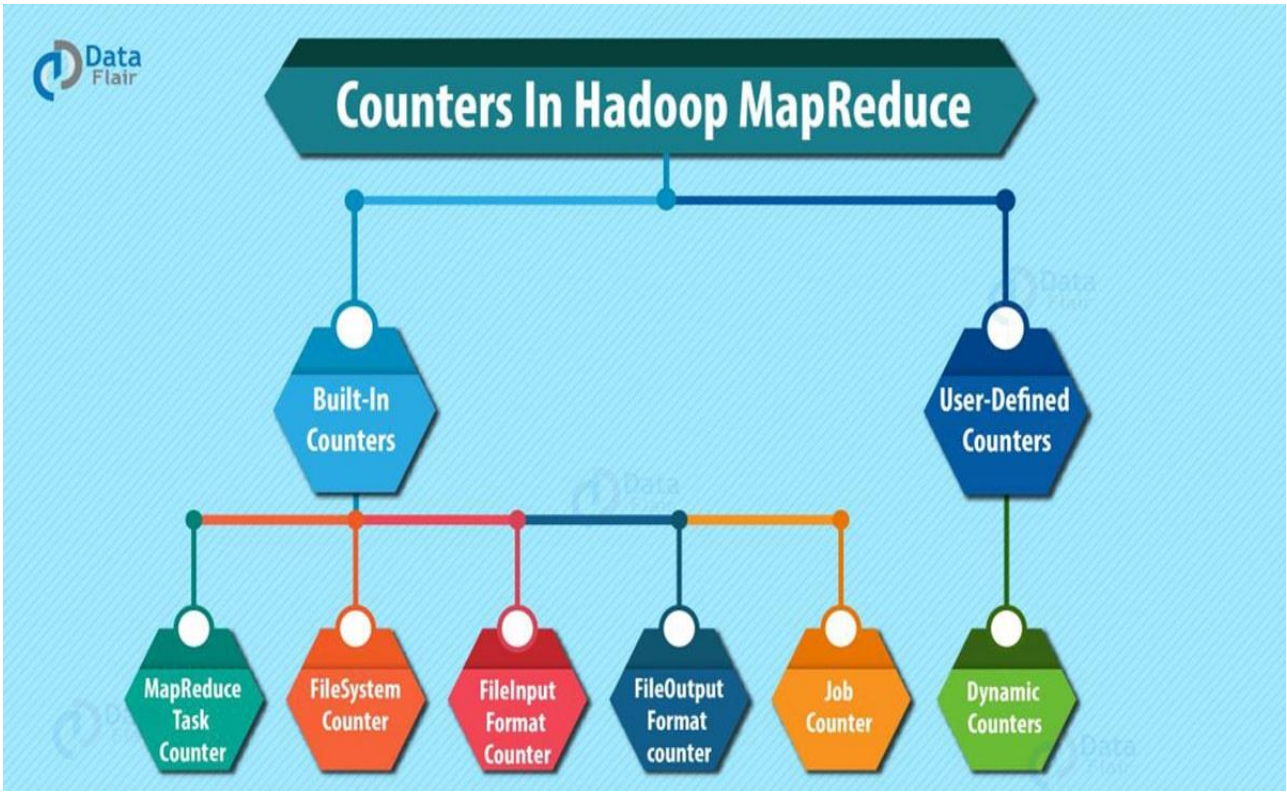
8. MapReduce Features (12M)

Ans:

Some of the more advanced features of MapReduce, including

- **Counters.**
- **Sorting.**
- **joining datasets.**
- **Side Data Distribution**
- **MapReduce Library Classes**

Hadoop Counters or MapReduce **Counters**



Hadoop Counters

MapReduce Framework has certain elements such as **Counters**, **Combiners**, and **Partitioners,** which play a key role in improving the performance of data processing. **Hadoop Counters:** Hadoop Counters provides a way to measure the progress or the number of operations that occur within **map/reduce job**. Counters in Hadoop MapReduce are a useful channel for gathering statistics about the MapReduce job: for quality control or for application-level. They are also useful for problem diagnosis.

Counters represent Hadoop global counters, defined either by the MapReduce framework or applications. Each Hadoop counter is named by an "**Enum**" and has a long for the value. Counters are bunched into groups, each comprising of counters from a particular Enum class.

Hadoop Counters validate that:

- The correct number of bytes was read and written.
- The correct number of tasks was launched and successfully ran.
- The amount of CPU and memory consumed is appropriate for our job and cluster nodes.

Read about Key – Value Pairs

Types of Hadoop MapReduce Counters

There are basically 2 types of MapReduce Counters:

- Built-In Counters in MapReduce
- User-Defined Counters/Custom counters in MapReduce

1. Built-In Counters in MapReduce

Hadoop maintains some built-in Hadoop counters for every job and these report various metrics, like, there are counters for the number of bytes and records, which allow us to confirm that the expected amount of input is consumed and the expected amount of output is produced.

Hadoop Counters are divided into groups and there are several groups for the built-in counters. Each group either contains task counters (which are updated as task progress) or job counter (which are updated as a job progress).

There are several groups for the Hadoop built-in Counters:

a. MapReduce Task Counter in Hadoop

Hadoop Task counter collects specific information (like number of records read and written) about tasks during its execution time. For example, the MAP_INPUT_RECORDS counter is the Task Counter which counts the input records read by each map task.

Hadoop Task counters are maintained by each task attempt and periodically sent to the application master so they can be globally aggregated.

b. FileSystem Counters

Hadoop FileSystem Counters in Hadoop MapReduce gather information like a number of bytes read and written by the file system. Below are the name and description of the file system counters:

- **FileSystem bytes read**– The number of bytes read by the filesystem by map and reduce tasks.
- **FileSystem bytes written**– The number of bytes written to the filesystem by map and reduce tasks.

c. FileInputFormat Counters in Hadoop

FileInputFormat Counters in Hadoop MapReduce gather information of a number of bytes read by map tasks via FileInputFormat. Refer this guide to learn about InputFormat in Hadoop MapReduce.

d. FileOutputFormat counters in MapReduce

FileOutputFormat counters in Hadoop MapReduce gathers information of a number of bytes written by map tasks (for map-only jobs) or reduce tasks via FileOutputFormat. refer this guide to learn about OutputFormat in Hadoop MapReduce in detail.

e. MapReduce Job Counters

MapReduce Job counter measures the job-level statistics, not values that change while a task is running. For example, TOTAL_LAUNCHED_MAPS, count the number of map tasks that were launched over the course of a job (including tasks that failed). Application master maintains MapReduce Job counters, so these Hadoop Counters don't need to be sent across the network, unlike all other counters, including user-defined ones.

2. User-Defined Counters/Custom Counters in Hadoop MapReduce

In addition to MapReduce built-in counters, MapReduce allows user code to define a set of counters, which are then incremented as desired in the **mapper** or **reducer.** For example, in Java, 'enum' is used to define counters.

A job may define an arbitrary number of 'enums', each with an arbitrary number of fields. The name of the enum is the group name, and the enum's fields are the counter names.

a. Dynamic Counters in Hadoop MapReduce

Java enum's fields are defined at compile time, so we cannot create new counters in Hadoop MapReduce at runtime using enums. To do so, we use dynamic counters in Hadoop MapReduce, one that is not defined at compile time using java enum.

Counters check whether the correct number of bytes is read or written, the correct number of tasks are launched and successfully run. Hence, Hadoop maintains built-in counters and user-defined counters to measure the progress that occurs within MapReduce job.

Joins

- The join operation is used to combine two or more database tables based on foreign keys.
- In general, companies maintain separate tables for the customer and the transaction records in their database. And, many times these companies need to generate analytic reports using the data present in such separate tables.
- Therefore, they perform a join operation on these separate tables using a common column (foreign key), like customer id, etc., to generate a combined table. Then, they analyse this combined table to get the desired analytic reports.

Joins in MapReduce

- Just like SQL join, we can also perform join operations in MapReduce on different data sets. There are two types of join operations in MapReduce:
- **Map Side Join:** As the name implies, the join operation is performed in the map phase itself. Therefore, in the map side join, the mapper performs the join and it is mandatory that the input to each map is partitioned and sorted according to the keys.
- **Reduce Side Join:** As the name suggests, in the reduce side join, the reducer is responsible for performing the join operation. It is comparatively simple and easier to implement than the map side join as the sorting and shuffling phase sends the values having identical keys to the same reducer and therefore, by default, the data is organized for us.

Steps required for Join

- Mapper reads the input data which are to be combined based on common column or join key.
- The mapper processes the input and adds a tag to the input to distinguish the input belonging from different sources or data sets or databases.
- The mapper outputs the intermediate key-value pair where the key is nothing but the join key.
- After the sorting and shuffling phase, a key and the list of values is generated for the reducer.
- Now, the reducer joins the values present in the list with the key to give the final aggregated output.

Side-Data Distribution

- Side data can be defined as extra read-only data needed by a job to process the main dataset.
- The challenge is to make side data available to the entire map or reduce tasks (which are spread across the cluster) in a convenient and efficient fashion.
- It is possible to cache side-data in memory in a static field, so that tasks of the same job that run in succession on the same task-tracker can share the data.
- “Task JVM Reuse” describes how to enable this feature. If you take this approach, be aware of the amount of memory that you are using, as it might affect the memory needed by the shuffle.

Using the Job Configuration

You can set arbitrary key-value pairs in the job configuration using the various setter methods on JobConf (inherited from Configuration). This is very useful if you need to pass a small piece of metadata to your tasks.

To retrieve the values in the task, override the configure() method in the Mapper or Reducer and use a getter method on the JobConf object passed in.

Usually, a primitive type is sufficient to encode your metadata, but for arbitrary objects you can either handle the serialization yourself (if you have an existing mechanism for turning objects to strings and back), or you can use Hadoop's Stringifier class.

- You shouldn't use this mechanism for transferring more than a few kilobytes of data because it can put pressure on the memory usage in the Hadoop daemons, particularly in a system running hundreds of jobs.
- The job configuration is read by the jobtracker, the tasktracker, and the child JVM, and each time the configuration is read, all of its entries are read into memory, even if they are not used.
- User properties are not read on the jobtracker or the tasktracker, so they just waste time and memory.

Distributed Cache

Rather than serializing side data in the job configuration, it is preferable to distribute datasets using Hadoop's distributed cache mechanism.

This provides a service for copying files and archives to the task nodes in time for the tasks to use them when they run. To save network bandwidth, files are normally copied to any particular node once per job.

MapReduce Library Classes

The classes and their methods that are involved in the operations of MapReduce programming.

- **JobContext Interface**
- **Job Class**
- **Mapper Class**
- **Reducer Class**

The JobContext interface is the super interface for all the classes, which defines different jobs in MapReduce. It gives you a read-only view of the job that is provided to the tasks while they are running.

S.No.	Subinterface Description
1.	MapContext<KEYIN, VALUEIN, KEYOUT, VALUEOUT>Defines the context that is given to the Mapper.
2.	ReduceContext<KEYIN, VALUEIN, KEYOUT, VALUEOUT>Defines the context that is passed to the Reducer.

Job Class

- Job class is the main class that implements the JobContext interface.
- The Job class is the most important class in the MapReduce API. It allows the user to configure the job, submit it, control its execution, and query the state.
- the user creates the application, describes the various facets of the job, and then submits the job and monitors its progress.

Here is an example of how to submit a job

```
// Create a new Job
Job job = new Job(new Configuration());
job.setJarByClass(MyJob.class);
// Specify various job-specific parameters
job.setJobName("myjob");
job.setInputPath(new Path("in"));
job.setOutputPath(new Path("out"));
job.setMapperClass(MyJob.MyMapper.class);
job.setReducerClass(MyJob.MyReducer.class);
// Submit the job, then poll for progress until the job is complete
job.waitForCompletion(true);
```

Constructors

S.No	Constructor Summary
1	**Job()**
2	**Job(Configuration conf)**
3	**Job(Configuration conf, StringjobName)**

Methods

S.No	Method Description
1	**getJobName() : User-specified job name.**
2	**getJobState() : Returns the current state of the Job.**
3	**isComplete() : Checks if the job is finished or not.**
4	**setInputFormatClass() : Sets the InputFormat for the job.**
5	**setJobName(String name) : Sets the user-specified job name.**
6	**setOutputFormatClass() : Sets the Output Format for the job.**
7	**setMapperClass(Class) : Sets the Mapper for the job.**
8	**setReducerClass(Class) : Sets the Reducer for the job.**
9	**setPartitionerClass(Class) : Sets the Partitioner for the job.**
10	**setCombinerClass(Class) : Sets the Combiner for the job.**

Mapper Class

he Mapper class defines the Map job. Maps input key-value pairs to a set of intermediate key-value pairs. Maps are the individual tasks that transform the input records into intermediate records. The transformed intermediate records need not be of the same type as the input records. A given input pair may map to zero or many output pairs.

Method

map is the most prominent method of the Mapper class. The syntax is defined below −
map(KEYIN key, VALUEIN value, Context context)
This method is called once for each key-value pair in the input split.

Reducer Class

The Reducer class defines the Reduce job in MapReduce. It reduces a set of intermediate values that share a key to a smaller set of values. Reducer implementations can access the Configuration for a job via the JobContext.getConfiguration() method. A Reducer has three primary phases − Shuffle, Sort, and Reduce.

- **Shuffle** − The Reducer copies the sorted output from each Mapper using HTTP across the network.

- **Sort** − The framework merge-sorts the Reducer inputs by keys (since different Mappers may have output the same key). The shuffle and sort phases occur simultaneously, i.e., while outputs are being fetched, they are merged.

- **Reduce** − In this phase the reduce (Object, Iterable, Context) method is called for each <key, (collection of values)> in the sorted inputs.

Method

reduce is the most prominent method of the Reducer class.

The syntax is defined below

reduce(KEYIN key, Iterable<VALUEIN> values, Context context)

This method is called once for each key on the collection of key-value pairs.